**컴퓨터 구조론 과제**

전자공학과 12181539 조권호

단일 사이클 설계는 비효율성 때문에 현대적 설계에서는 쓰이지 않는다. 단일 사이클 설계에서는 클럭 사이클이 모든 명령어에 대해 같은 길이를 가져야 하기 때문이다. CPI 값은 1이지만 클럭 사이클이 너무 길기 때문에 전체 성능이 좋지 않다. 또한 부동 소수점 유닛을 구현하려 하거나 좀 더 복잡한 명령어를 갖는 명령어 집합인 경우에는 잘 작동하지 않을 것이다. 클럭 사이클은 모든 명령어에 대한 최악의 지연과 같다고 가정해야 하기 때문에 자주 생기는 경우의 지연은 줄여 주지만 최악의 경우 사이클 시간을 개선하지 못하는 구현은 소용이 없다. 따라서 단일 사이클 구현은 자주 생기는 일을 빠르게 라는 핵심 설계 원칙을 위반한다.

멀티 사이클 이용

접근: 실행을 완료하는 데 1회 이상의 클럭 사이클이 소요되도록 한다. 실행을 각 사이클이 하나의 주요 기능 유닛만 사용하도록 제한하도록 나눈다. 모든 명령어가 동일한 수의 클럭 사이클을 사용하는 것은 아니다. 사이클당 하나의 메모리 액세스만 가능. 명령어와 데이터 모두 하나의 메모리만 필요. ALU가 오직 1개만 필요. 사이클당 하나의 ALU 작업만 수행.

실행: 각 사이클이 끝날 때 임시 레지스터에 현재 명령에 의해 다음 주기에 필요한 값을 저장한다. 모든 데이터(IR 제외)는 인접한 클럭 사이클 쌍 사이에서만 데이터를 보유한다(쓰기 제어 신호 필요 없음). 후속 명령에 사용되는 데이터는 프로그래머 가시 레지스터에 저장된다. (임시 레지스터: IR, A, B, ALUOut, MDR)

멀티 사이클에서는 단일 사이클에서보다 3개의 멀티플렉서를 더 추가하는데 메모리 접근에 대한 멀티플렉서, A이 대한 ALU input을 위한 멀티플렉서, B에 대한 ALU input을 위한 멀티플렉서가 추가로 필요하다. 이때 PC Update logic 멀티 플렉서가 output of ALU, ALUOut, PC[31..28]:IR[25..0][00]for jump에 대해 나눠진다. 또한 write control for PC에 대한 값이 PC Write + (PCWriteCond⋅ALUzero)에 대해 결정된다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| IF | ID | EX | MEM | WB |

1cycle 1cycle 1cycle 1cycle 1cycle

IF: PC 가져오기 및 업데이트(공통)

ID: 명령 디코드, 레지스터 읽기, 부호 확장 오프셋(공통)

EX: R-type 실행; 메모리 주소 계산; 분기 비교; 분기 및 점프 완료

MEM: 메모리 읽기, 메모리 쓰기 실행

WB: 로드 실행

클럭 사이클 효율적 사용가능-> 명령어 별로 필요한 만큼만 점유 가능-> waste time 메울 수 있음

하나의 명령어 실행에서 각각의 functional unit 들이 여러 번 사용 가능 -> cycle로 분리되기 때문 -> temporart register 사용해야함, mux를 더 사용해야한다.

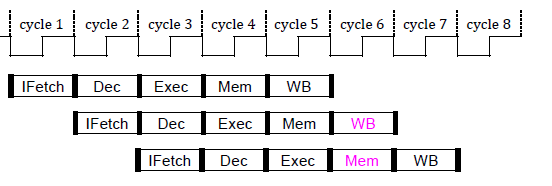
lw를 제외한 모든 명령어들에 대해서는 실행시간에서 무조건 이득.

파이프라인 이용

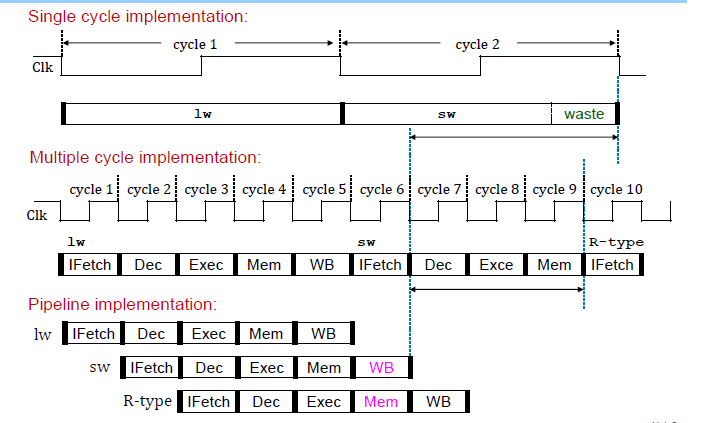
접근: 더 많은 사이클을 실행을 만드는 것은 속도를 더 빠르게 만들 수 있다. 모든 최신 프로세서는 파이프 라이닝을 사용  
파이프 라인을 사용하면 각각의 실행은 대략 같은 시간이 걸리고 동시에 여러 명령의 중복 실행(다른 명령이 동시에 활성화)한다. 파이프라인은 명령 처리량을 증가시켜 성능을 향상시킴(개별 명령의 실행 시간을 줄이지는 않는다) 파이프라인은 단일 작업의 대기 시간이 아니라 전체 워크로드의 처리량을 지원함(여러 작업을 동시에 작업시킴). 가장 느린 파이프라인 단계에 따라 파이프라인 속도가 제한됨

실행: 파이프라인이 꽉 차면 사이클마다 하나의 명령이 완료되므로 CPI = 1

IF: 명령 가져오기  
ID/RF: 명령 디코딩 및 레지스터 파일 읽기  
EX: 실행 또는 주소 계산  
MEM: 데이터 메모리 액세스  
WB: 다시 쓰기



현재 실행이 끝나기 전에 다음 실행 시작 -> 처리량 향상, instruction latency는 감소하지 않음.



* 파이프라인을 통해 더 빠르게 명령들을 처리할 수 있다.

이러한 구현은 MIPS DATAPATH에서 각 단계사이를 파이프라인 레지스터를 이용해 분리한다.

이러한 방법은 MIPS에서 모든 명령이 동일한 길이(32비트), 첫 번째 단계에서 가져오고 두 번째 단계에서 디코딩, 여러 형식에 걸쳐 대칭성을 갖는 소수의 명령 형식(3개), 두 번째 단계에서 레지스터 파일 읽기를 시작, 메모리 작업은 로드 및 저장소에서만 발생, 실행 단계에서 메모리 주소를 계산 가능, 각 MIPS 명령은 최대 한 개의 결과(즉, 시스템 상태 변경)를 기록하고 파이프라인의 끝(MEM 및 WB) 근처에서 기록하는 특성으로 쉽게 구현이 가능하지만 structural hazards, data hazards, control hazards 등이 구현을 어렵게 한다. Bypassing(파이프라인 레지스터에 있는 가장 이른 지점에서 결과를 가져와 해당 사이클에서 결과를 필요로 하는 기능 단위(예: ALU)로 전달, ALU 기능 장치의 경우 입력은 ID/EX가 아닌 모든 파이프라인 레지스터에서 가져올 수 있다, ALU의 입력에 멀티플렉서 추가, EX/MEM 또는 MEM/WB의 Rd 쓰기 데이터를 EX 단계의 Rs 및 Rt ALU 입력 중 하나(또는 둘 다)에 연결, 새 mux를 제어하기 위한 적절한 제어 하드웨어 추가, 다른 기능 장치에도 유사한 전달 로직(예: DM)이 필요할 수 있음, 포워딩을 통해 데이터 의존성이 있는 경우에도 CPI 1을 달성 가능, 전달을 위해 멀티플렉서를 추가하면 설계의 중요한 경로가 증가, 신중하게 설계될 필요가 있음)

Structural hazards: 리소스 충돌(두 명령이 동일한 하드웨어를 사용하려고 할 때 발생, 일부 기능 단위가 완전히 파이프 라인 되지 않을 때 종종 발생)

해결 - 보다 복잡한 기능 단위를 구축, 딜레이 시킴, 단위가 완전히 파이프라인 되지 않은 경우 mult, div 사용, 하드웨어가 파이프라인 stall을 삽입하여 위반되는 명령을 지연, 이상적인 값 1에서 CPI를 증가시킴, 레지스터를 사용할 때 사이클 후반부에 읽고 전반부에 쓰기를 수행하여 레지스터 파일 액세스 위험 해결, 전방전달(forwarding) 또는 우회전달(bypassing) 사용

Data hazards: 어떤 단계가 다른 단계가 끝나기를 기다려야 하기 때문에 파이프라인이 지연되어야하는 경우 발생한다. 쓰기 후 읽기(RAW), 쓰기 후 쓰기(WAW), 읽은 후 쓰기(WAR) -WAW와 WAR은 renaming을 통해 해결할 수 있다.

해결- 소프트웨어에 NOP를 삽입, 파이프라인에 bubble을 삽입하기 위한 하드웨어를 만듦, 파이프라인 내에서 데이터 전달 사용(파이프라인 레지스터)

Control hazards: 다른 명령어들이 실행되는 동안 어떤 명령어의 결과에 기반을 둔 결정을 할 필요가 있을 때 발생한다. branch instruction, jump instruction

해결- stall( 주소를 알 때까지 기다린다, 그러나 CPI를 증가시킨다, 성능 저하), 의사 결정 지점을 가능한 한 조기에 이동(추가 하드웨어 비용으로 중단 횟수 감소), 지연된 분기(컴파일러 지원)(두 개 이상의 지연 슬롯을 채워야 하는 더 깊은 파이프의 경우 실현 불가능), 안사용된다고 예측(명령어를 계속 가져오고 실행, 예측이 틀렸다면 지시를 무효화할 수 있어야 함, 까다로울 수 있지만 정확히 한다면 그렇게 어렵지 않음), 사용된다고 예측(대상을 계산해야 하므로 더 복잡, 일반적으로 이 작업은 시간이 좀 걸림, 저장 대상 주소 예측, 틀렸을 때 여전히 무효화할 필요가 있음, 대부분의 시스템이 어떤 유형의 예측을 수행함(실행됨, 비실행됨)) -> 예측 (더 많은 하드웨어를 사용하면 분기 예측(BPB)이 올바르고 분기별 주소 및/또는 명령이 캐시(BTB)된 경우 제어 위험 지연의 영향을 훨씬 더 줄일 수 있음)